Variance – What is it?

- Most of the modern progamming languages support **subtyping** which allows us to implement hierarchies like **a Cat Is-An creature**.
- In **Java** we use **extend** to *exchange/expand* the behaviour of an **existing class** or use **implements** to *provide* an implementation for an **interface.**
- The word **variance** in this context is used to describe how **subtyping in** complex aspects like:
  - **Method return type**
  - **Type declaration**
  - **Arrays relates to the direction of inheritance**

  of the involved classes.

Variance – Covariance in Java

```java
public class Creature {}
```

```java
public class Insect extends Creature {}
```

In **Java** a **overriding method** needs to be **covoriant in it‘s return type**

```java
abstract public class Veterinary {
    abstract public Creature treat();
}
```

```java
public class InsectVeterinary extends Veterinary {
    @Override
    public Insect treat() {
        return null;
    }
}
```

Subclasses can be **cast up the inheritance tree**, while **downcasting** will **cause an error**

```java
(Insect) new Creature()
```

// Error

```java
(Creature) new Insect()
```

Variance – Covariance in Java

In **Java arrays** are **covariant**.

But, there is a big problem with this:

```
Integer[] numbers = {1,2,3,4};
Object[] objects = numbers;
objects[0] = "String";
```

// Runtime exception: Exception in thread “main”...

Variance – Generic Collections

- As of **Java 1.5** we can use **Generics** in order to tell the compiler which elments are supposed to be stored in our collections. Unlike Arrays, **generic collections** are **invariant** in their parameterized type **by default**.
- This means you can‘t substitute List<Creature> with List<Insect>, it won‘t even compile.
- Fortunately, the *user can specify the variance* of *type parameters* himself *when using generics*, which is called **use-site-variance.**

Variance – Covariant collections In Java

The following code snippet shows how to declare a **covariant** list of Creatures and assign a list of insect to it.

```
List<Insect> insects = new ArrayList<>();
List<? extends Creature>  creatures = insects;
```

Such a **covariant list** still differs from a an array, because the covariance is encoded in it‘s type parameter, which means we can **only** read from it.

```
creatures.add(new Insect());          // Won‘t compile
```

Variance – Contravariant collections In Java

The following code snippet shows how to declare a contravariant list of creatures.

```
List<Creature> creatures = new ArrayList<>();
List<? super Creature>  contraVariantCreatures = creatures;
```

Like with covariant lists, we don‘t know for sure which type the list contains. The difference is, **we can‘t read from** it, since it is unclear if we’ll get a creature or just a plain object. But know we **can write to it**, as we know that at least a creature may be added.

```
contraVariantCreatures.add(new Insect());
```

```
Creature creature = contravariantCreatures.get(0);      // Won‘t compile
```

Variance – Summary of Java Variance

Joshua Bloch created a rule of thumb in is book **Effective Java**:

*"Producer – extends, consumer – super (**PECS**)"*

Covariant   Contravariant

Read   Write

Variance – of type collections in Kotlin

Kotlin is different from Java regarding generics and also arrays.

- The easiest difference is that **arrays** in **Kotlin** are **invariant**

```
var stringArray: Array<String> = arrayOf()

var objectArray: Array<Object> = stringArray        // Won‘t compile
```

But, is there a way to work safely with subtyped arrays ?

Variance – Type collections in Kotlin

As you have seen, Java uses the „wildcard types“ to make generics variant.

Which is called ***use-site-variance***

In Kotlin we use ***declaration-site-variance***

The generic parameter T in Kotlin can be marked as „only produced“ with the **out** keyword, which makes T **covariant** or can be marked as „only- onsumed“ with the **in** keyword, which makes T **contravariant**.

*“Consumer in, producer out (**CIPO**)*

Variance – Covariance in Kotlin

The problem with covariance is with the mutablity after upcasting. Covariant type parameters - not only setters, but on any **in** position (*public mehtod parameters or public properties*) are a potential source of errors.

This is why Kotlin prohibits covariant type parameters used on **in** positions.

```
class Container<out T> constructor(var element: T){
    fun set(new: T) {
        element = new
    }
    fun get(): T = element
}
```

```
class Container<out T> constructor(private var element: T){

    fun get(): T = element
}
```

Variance – Contravariance in Kotlin

You might guess that contravariant parameters, which are made using the **in** keyword are only allowed on **out** positions.

```
class Container<in T> constructor(var element: T){
    fun set(element: T){
        this.element = element
    }
    fun get(): T = element
}
```

```
class Container<in T> constructor(private var element: T){

    fun set(element: T){
        this.element = element
    }
}
```

Variance – Type projections in Kotlin

Sadly it is not always enough to have the opportunity of declaring a type parameter **T** as either **in** or **out**.

As an alternative Kotlin also allows sort of „use-site-variance“ which is called **type-projection**.

```
class Container<T> constructor(private var element: T){

   fun copy(from: Array<out T>, to: Array<T>){
      // ...
   }

   fun fill(dest: Array<in T>, value: T){
      // ...
   }
}
```

End of section: Variance

Any questions ?